Swami Sahajanand College of Computer Science

B.C.A Sem-V

Subject – Software Engineering

UNIT - 2

Software Planning & Designing

- Team Structure – Egoless team, Chief Programmer Team, Controlled Decentralized Team
- System Design principles.
- Module level concepts - Coupling & Cohesion
- Design Methodology - Structure Chart
- Functional approach vs. Object Oriented Approach

Software Project Team Organization

There are many ways to organize the project team. Some important ways are as follows:

1. Hierarchical team organization
2. Chief-programmer team organization
3. Matrix team, organization
4. Egoless team organization
5. Democratic team organization

1.Hierarchical team organization

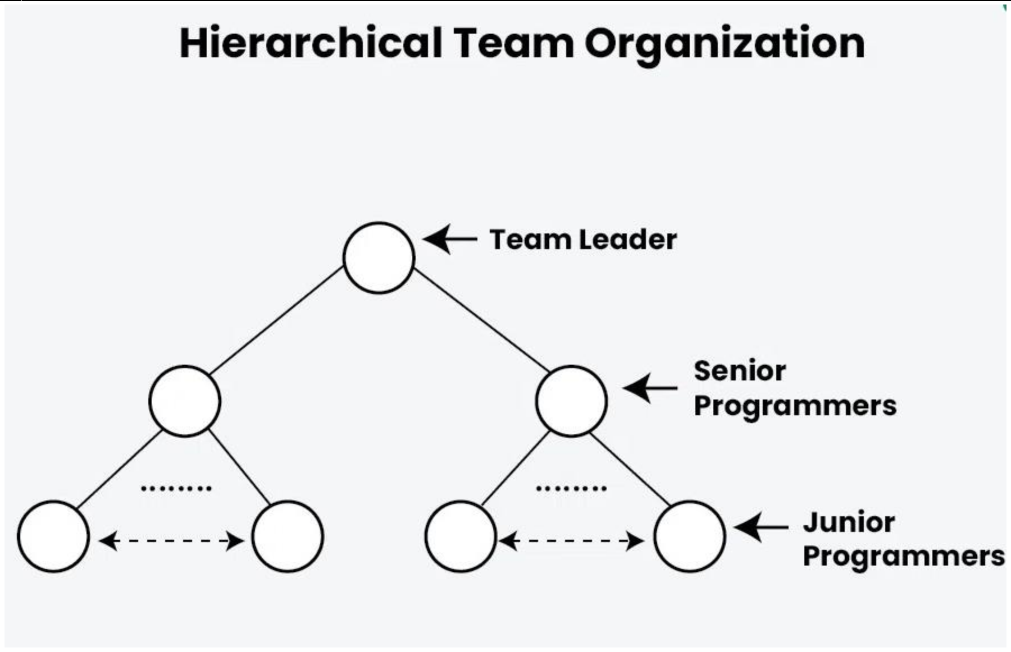
In this, the people of the organization at different levels follow a tree structure. People at the bottom level generally possess the most detailed knowledge about the system. People at higher levels have a broader appreciation of the whole project.

Below are some benefits of Hierarchical Team Organization:

- It limits the number of communications paths and still allows for the needed communication.
- It can be expanded over multiple levels.
- It is well suited for the development of hierarchical software products.
- Large software projects may have several levels.

Below are some Limitations of hierarchical team organization:

- As information has to travel up the levels, it may get distorted.
- Levels in the hierarchy often judge people socially and financially.
- Most technically competent programmers tend to be promoted to management positions which may result in the loss of good programmers and also bad managers.



2. Chief-programmer team organization

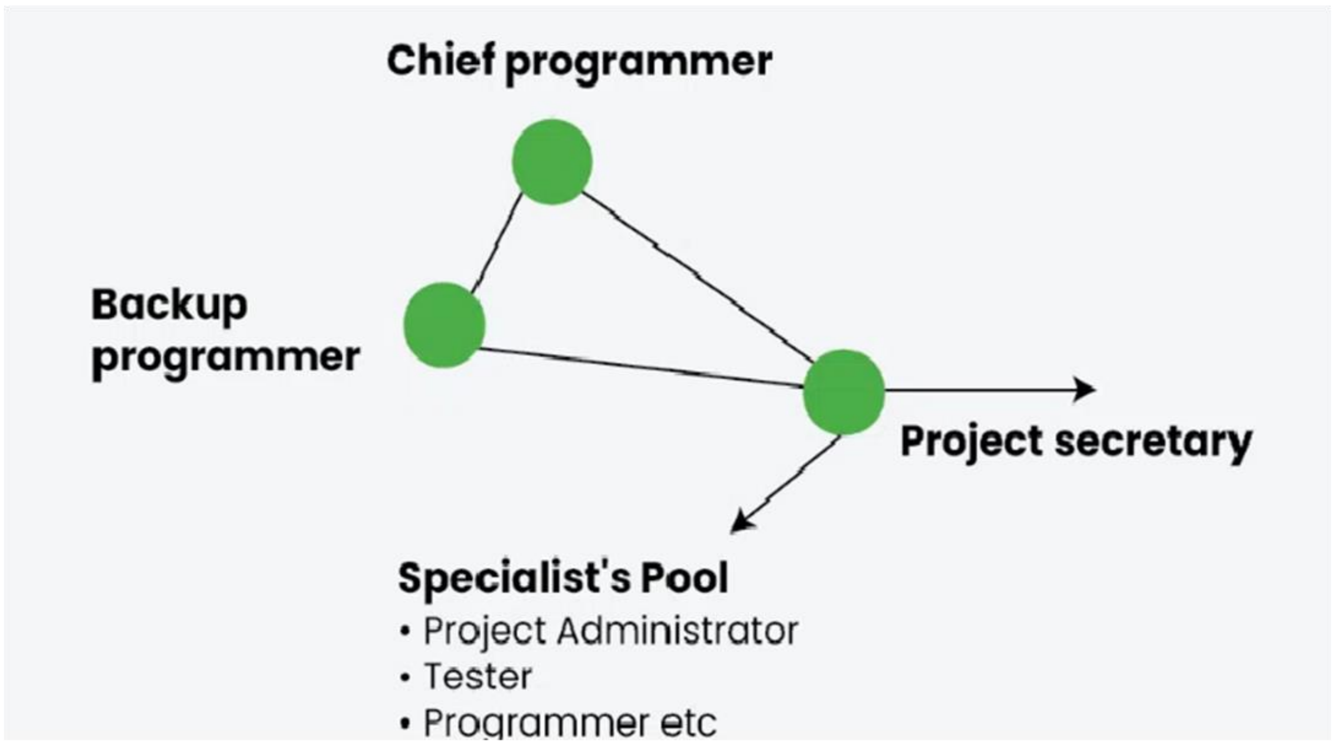
The chief programmer team organization is composed of a small team consisting of the following team members

•**The Chief programmer** is the person who is actively involved in the planning, specification and design process and ideally in the implementation process.

•**The project assistant** is the closest technical co-worker of the chief programmer.

•The project secretary relieves the chief programmer and all other programmers of administration tools.

- **Specialists:** These people select the implementation language, implement individual system components, employ software tools, and carry out tasks.



Chief-programmer team organization

Advantages of Chief-programmer team organization

- Centralized decision-making
- Reduced communication paths
- Small teams are more productive than large teams
- The chief programmer is directly involved in system development and can exercise better control functions.

Disadvantages of Chief-programmer team organization:

- Project survival depends on one person only.
- This can cause psychological problems as the "chief programmer" is like the "king" who takes all the credit and other members are resentful.
- Team organization is limited to only a small team and a small team cannot handle every project.
- The effectiveness of the team is very sensitive to the Chief programmer's technical and managerial activities.

<u>3. Egoless Team Organization</u>

Egoless programming is a state of mind in which a programmer is supposed to separate themselves from their product.In this team organization goals are set and decisions are made by group consensus. Here the group, 'leadership' rotates based on tasks to be performed and differing abilities of members.

In this organization work products are discussed openly and freely examined by all team members. There is a major risk with such an organization if teams are composed of inexperienced or incompetent members.

4.Matrix Team Organization

In a matrix team organization, team members are grouped based on their specialties, such as developers, testers, or designers. Each specialist group is led by a Project manager who oversees their work. This structure allows team members to focus on their areas of expertise while managers coordinate tasks and resources across different projects.

5. Democratic Team Organization

It is quite similar to the egoless team organization, but one member is the team leader with some responsibilities:

- Coordination
- Final decisions, when consensus cannot be reached.

Advantages of Democratic Team Organization

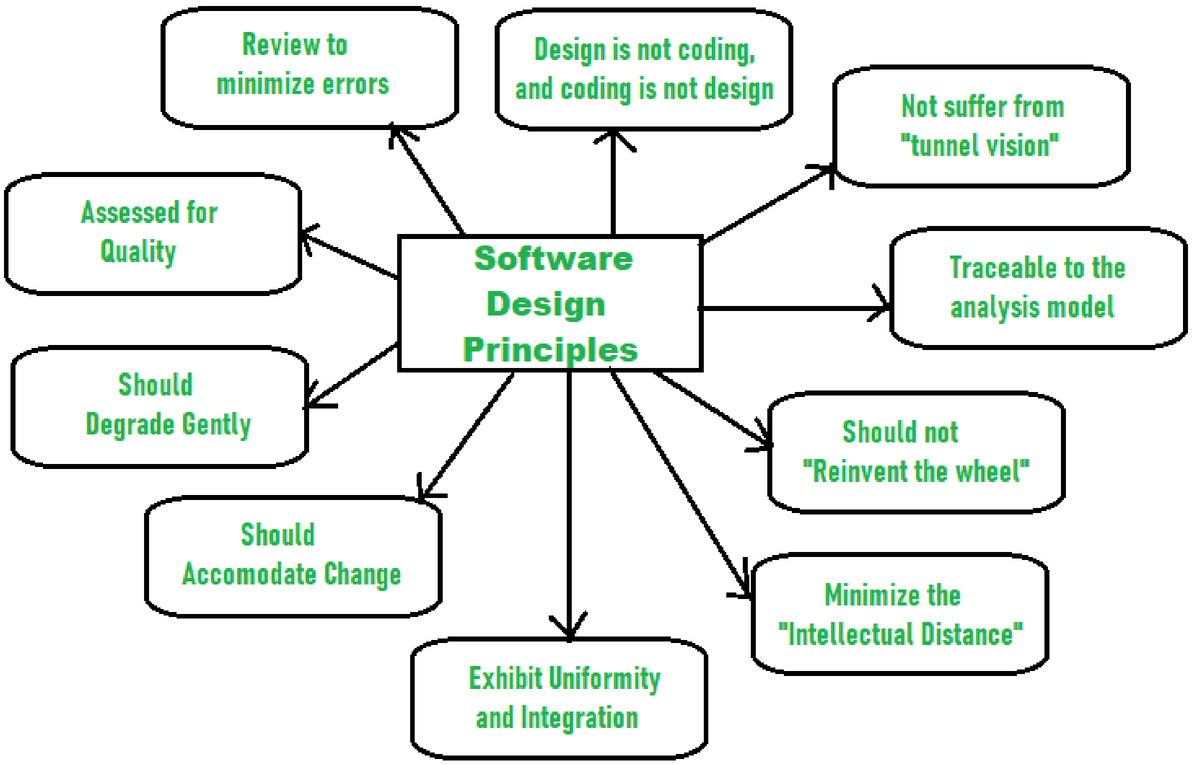
- Each member can contribute to decisions.
- Members can learn from each other.
- Improved job satisfaction.

Disadvantages of Democratic Team Organization

- Communication overhead increased.
- Need for compatibility of members.
- Less individual responsibility and authority.

System Design Principles

Software Design is also a process to plan or convert the software requirements into a step that is needed to be carried out to develop a software system. There are several principles that are used to organize and arrange the structural components of Software design. Software Designs in which these principles are applied affect the content and the working process of the software from the beginning.


Software Design Principles
Review to minimize errors
Design is not coding, and coding is not design
Not suffer from "tunnel vision"
Assessed for Quality
Traceable to the analysis model
Should Degrade Gently
Should not "Reinvent the wheel"
Should Accomodate Change
Minimize the "Intellectual Distance"
Exhibit Uniformity and Integration

Principles of Software Design:

1. **Should not suffer from "Tunnel Vision" –**

   While designing the process, it should not suffer from "tunnel vision" which means that it should not only focus on completing or achieving the aim but on other effects also.

2. **Traceable to analysis model –**

   The design process should be traceable to the analysis model which means it should satisfy all the requirements that software requires to develop a high-quality product.

3. **Should not "Reinvent The Wheel" –**

   The design process should not reinvent the wheel, that means it should not waste time or effort in creating things that already exist. Due to this, the overall development will increase.

4. **Minimize Intellectual distance –**

   The design process should reduce the gap between real-world problems and software solutions for that problem meaning it should simply minimize intellectual distance.

5. **Exhibit uniformity and integration –**

   The design should display uniformity which means it should be uniform throughout the process without any change. Integration means it should mix or combine all parts of software i.e. subsystems into one system.

6. **Accommodate change –**

   The software should be designed in such a way that it accommodates the change implying that the software should adjust to the change that is required to be done as per the user's need.

7. **Degrade gently –**

   The software should be designed in such a way that it degrades gracefully which means it should work properly even if an error occurs during the execution.

8. **Assessed or quality –**

   The design should be assessed or evaluated for the quality meaning that during the evaluation, the quality of the design needs to be checked and focused on.

9. **Review to discover errors –**

   The design should be reviewed which means that the overall evaluation should be done to check if there is any error present or if it can be minimized.

10. **Design is not coding and coding is not design –**

Design means describing the logic of the program to solve any problem and coding is a type of language that is used for the implementation of a design.

Software Design principal

1. Problem Partitioning

For small problems, we can handle the entire problem at once but for the significant problem, divide the problems and conquer the problem. It means to divide the problem into smaller pieces so that each piece can be captured separately.

For software design, the goal is to divide the problem into manageable pieces.

Benefits of Problem Partitioning

1. Software is easy to understand
2. Software becomes simple
3. Software is easy to test
4. Software is easy to modify
5. Software is easy to maintain
6. Software is easy to expand

These pieces cannot be entirely independent of each other as they together form the system. They have to cooperate and communicate to solve the problem. This communication adds complexity.

2.Abstraction

An abstraction is a tool that enables a designer to consider a component at an abstract level without bothering about the internal details of the implementation. Abstraction can be used for existing elements as well as the component being designed.

Here, there are two common abstraction mechanisms

1. Functional Abstraction
2. Data Abstraction

Functional Abstraction

Functional abstraction specifies the functions that a module performs in the system

1. A module is specified by the method it performs.
2. The details of the algorithm to accomplish the functions are not visible to the user of the function.
3. Function prototype, function call, closed subroutine are some examples of functional abstraction.
4. Other programmers can use your abstraction by invoking the function. If you have done a good job of documenting the function, they don't have to read your code to use it. For example, you might declare a function search as follows: /** Return an index of x in a.

   Functional abstraction forms the basis for **Function oriented design approaches**.

Data Abstraction

1. Details of the data elements are not visible to the users of data.
2. Data abstraction specifies the entities or data objects that provide certain services to the external environment.
3. Abstract data types (ADTs), such as structures in C, classes in C++, and packages in Java are examples of data abstraction. A more detailed example for the ADT of stack in C++.
4. Data Abstraction forms the basis for **Object Oriented design approaches**.

Advantages of abstraction:

• It separates design from implementation, which is easy to understand and manage.
• It helps in problem understanding and software maintenance.
• It reduces the complexity of modern computer programming for software users and engineers.
• It helps in program organization that can be generalized for recovering common problems and therefore it promotes software reuse.
• It also promotes scalability and helps in making early design decisions

3.Modularity

Modularity in software engineering means breaking complex software systems down into smaller manageable modules or components that are tightly coupled together. They can also be constructed as independent subsystems designed and executed individually apart from other system elements since each module carries out a specific mission. The aim, therefore, is to simplify by modularizing the program into units or reusable building blocks that can be easily exchanged for one another.

Modularity specifies the division of software into separate modules which are differently named and addressed and are integrated later on in order to obtain the completely functional software. It is the only property that allows a program to be intellectually manageable. Single large programs are difficult to understand and read due to a large number of reference variables, control paths, global variables, etc.

- Each module is a well-defined system that can be used with other applications.
- Each module has single specified objectives.
- Modules can be separately compiled and saved in the library.
- Modules should be easier to use than to build.
- Modules are simpler from outside than inside.

4.Strategy of Design

A good system design strategy is to organize the program modules in such a method that are easy to develop and latter too, change. Structured design methods help developers to deal with the size and complexity of programs. Analysts generate instructions for the developers about how code should be composed and how pieces of code should fit together to form a program.

To design a system, there are two possible approaches:

1. Top-down Approach
2. Bottom-up Approach

❖ **Top Down Design**

We know that a system is composed of more than one sub-systems and it contains a number of components. Further, these subsystems and components may have their own set of sub-system and components and create hierarchical structure in the system.

Top-down design takes the whole software system as one entity and then decomposes it to achieve more than one sub-system or component based on some characteristics. Each sub-system or component is then treated as a system and decomposed further. This process keeps on running until the lowest level of the system in the top-down hierarchy is achieved. Top-down design starts with a generalized model of the system and keeps on defining the more specific part of it. When all components are composed the whole system comes into existence.

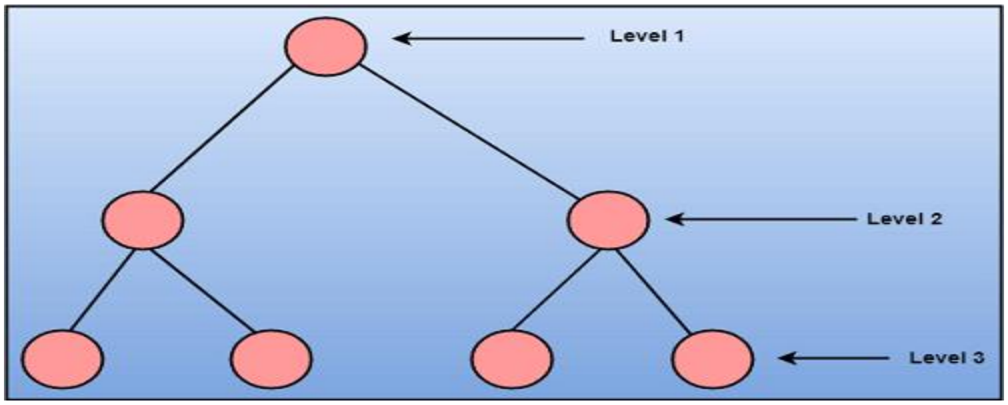
Examples:

1. **Educational Program Design**

A university might redesign its curriculum using a top-down approach. First, the educational goals and outcomes for a department are defined. Then, specific courses and content are developed to meet these predefined goals

2. **Software Development**

In software engineering, a top-down approach might involve defining the software's architecture and high-level functionality before breaking these down into modules and detailed coding tasks. For instance, a development team at Microsoft might start by defining the overall functionality for a new feature in Microsoft Office and then detailing the specific components and tasks required to build that feature.



❖ **Bottom-up Design**

The bottom up design model starts with most specific and basic components. It proceeds with composing higher-level components by using basic or lower level components. It keeps creating higher level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased.

Bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.

Whole Foods (now part of Amazon): Before its acquisition by Amazon, Whole Foods was known for allowing individual stores to decide what products to stock based on local customer preferences and feedback.

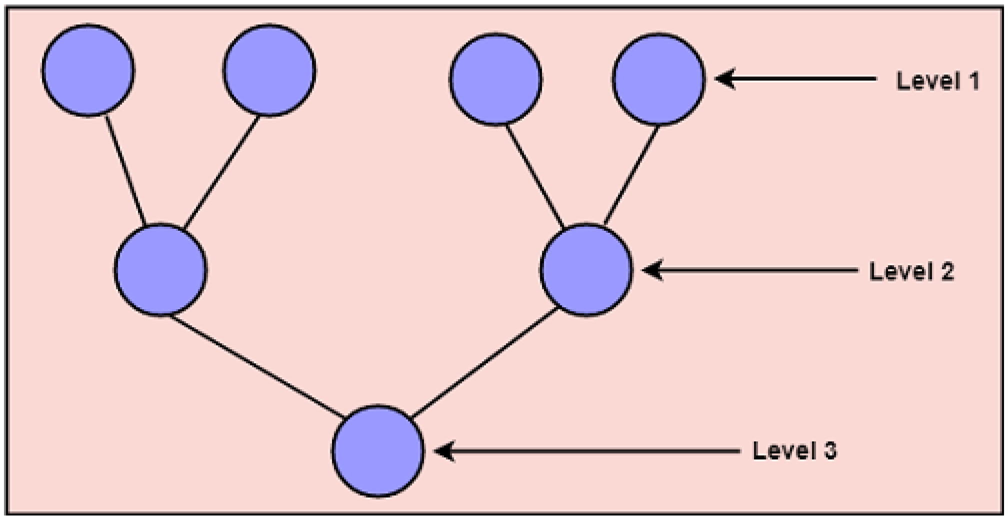
Examples:

Research and Development (R&D)

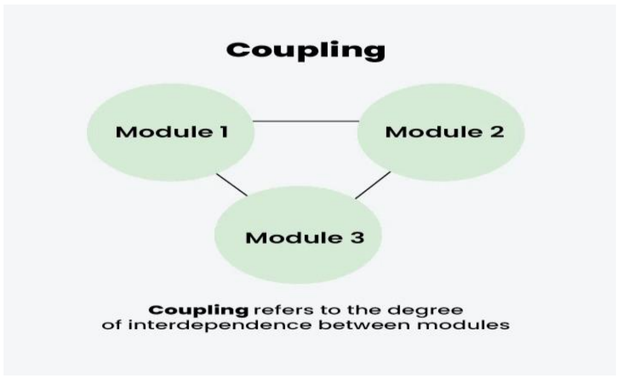
In an R&D setting, such as at a biotech firm, scientists and researchers might work independently or in small teams on specific experiments or studies based on their areas of expertise. The results from these individual projects contribute to the overall understanding of a larger research question, like developing a new pharmaceutical drug.

Software Bug Fixes and Enhancements

A software company might employ a bottom-up approach by encouraging developers to identify and fix bugs or enhancements based on user feedback. These improvements are incorporated into the next software release cycle, improving product quality.

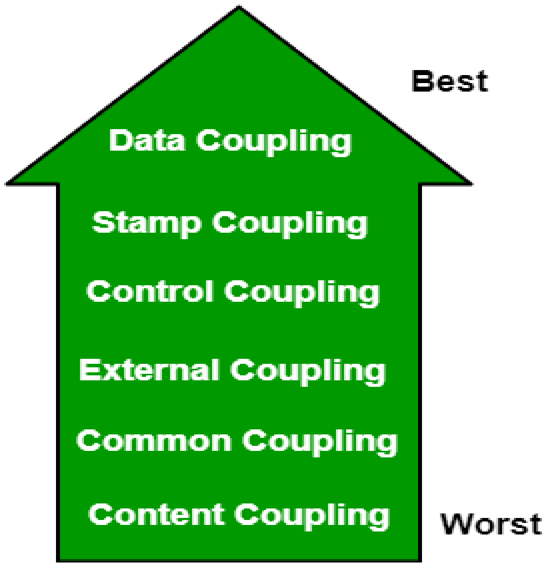


- **Module level concepts - Coupling & Cohesion**
  Coupling:
  Coupling refers to the degree of interdependence between software modules. High coupling means that modules are closely connected and changes in one module may affect other modules. Low coupling means that modules are independent, and changes in one module have little impact on other modules.


Coupling
Module 1
Module 2
Module 3
Coupling refers to the degree of interdependence between modules

Coupling is the measure of the degree of interdependence between the modules. A good software will have low coupling.

types of Coupling:



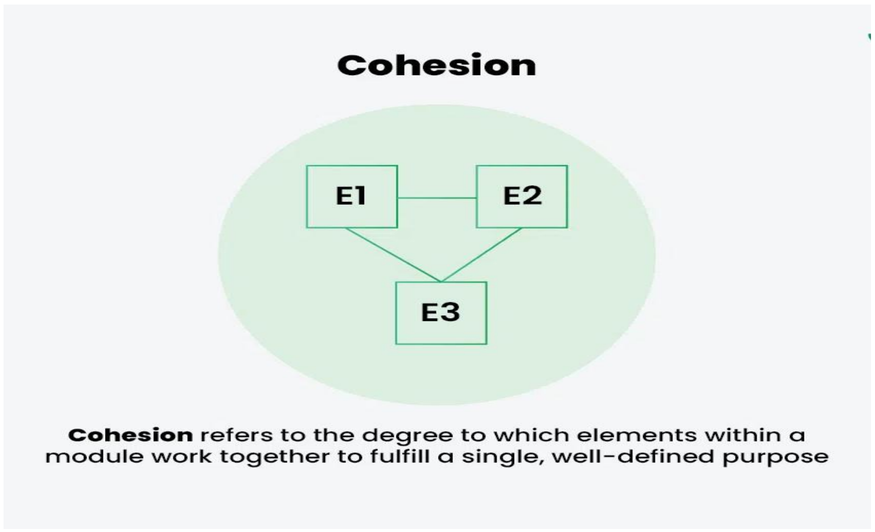
- **Data Coupling:** If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent of each other and communicate through data. Module communications don't contain tramp data. Example-customer billing system.
- **Stamp Coupling:** In stamp coupling, the complete data structure is passed from one module to another module. Therefore, it involves tramp data. It may be necessary due to efficiency factors- this choice was made by the insightful designer, not a lazy programmer.example:data structure
- **Control Coupling:** If the modules communicate by passing control information, then they are said to be control coupled. It can be bad if parameters indicate completely different behavior and good if parameters allow factoring and reuse of functionality. Example- sort function that takes comparison function as an argument.
- **External Coupling:** In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Ex-protocol, external file, device format, etc.
- **Common Coupling:** The modules have shared data such as global data structures. The changes in global data mean tracing back to all modules which access that

data to evaluate the effect of the change. So it has disadvantages like difficulty in reusing modules, reduced ability to control data accesses, and reduced maintainability.

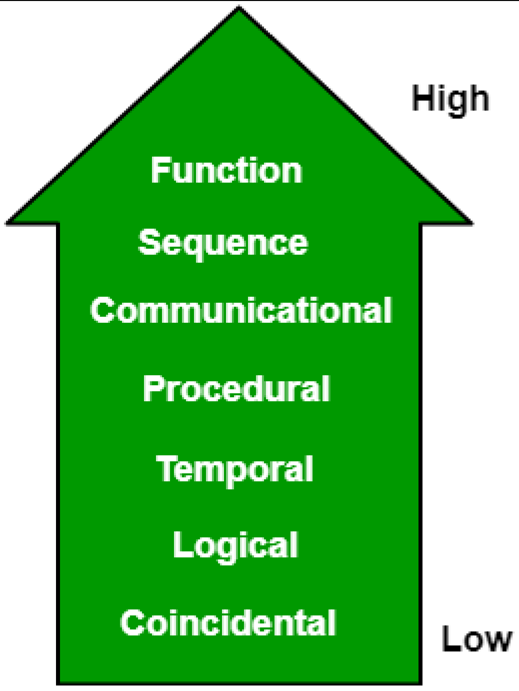
- **Content Coupling**: In a content coupling, one module can modify the data of another module, or control flow is passed from one module to the other module. This is the worst form of coupling and should be avoided.

❖ Cohesion:

Cohesion refers to the degree to which elements within a module work together to fulfill a single, well-defined purpose. High cohesion means that elements are closely related and focused on a single purpose, while low cohesion means that elements are loosely related and serve multiple purposes.


Cohesion
E1
E2
E3
Cohesion refers to the degree to which elements within a module work together to fulfill a single, well-defined purpose

Cohesion is a measure of the degree to which the elements of the module are functionally related. It is the degree to which all elements directed towards performing a single task are contained in the component. Basically, cohesion is the internal glue that keeps the module together. A good software design will have high cohesion.



Levels of cohesion

- **Functional Cohesion:** In a functionally cohesive module, all the elements of the module perform a single function. For example, "log" computes the logarithm of a number and "printf" prints the results.
- **Sequential Cohesion:** Sequential cohesion exists when the output from one element of a module becomes the input for some other element.. For example, "withdraw money" and "update balance" both are bound together to withdraw money from an account.
- **Communicational Cohesion:** In communicational cohesion, all the elements of a module operate on the same input or output data. For example, "print and punch the output file" can be communicational cohesion. The binding of elements in a module is higher than procedural cohesion.
- **Procedural Cohesion:** Procedural cohesion contains the elements which belong to a common procedural unit. The functions are executed in a certain order. For example, "entering, reading, and verifying the ATM password" are bound in an ordered manner for the procedurally cohesive module "enter password."
- **Temporal Cohesion:** Sometimes, a module performs several functions in a sequence but their execution is related to a certain time. For example, "a database trigger is activated on executing a certain procedure."

- **Logical Cohesion:** Logical cohesion exists when logically-related elements of a module are placed together. All the parts communicate with each other by passing control information such as flag variables, using some shared source code, etc.
- **Coincidental Cohesion:** It occurs when the elements within a given module have no meaningful relationship to each other

Design Methodology

- ➔ The aim of design methodologies is not to reduce the process of design to a sequence of mechanical steps to provide guidelines to the designer during the design process.
- ➔ Structure design methodology (SDM) views every software system as having some inputs that are converted into the desired outputs by the software system The software is viewed as a transformation function that transforms the given inputs into the desired outputs.
- ➔ The structured design methodology is primarily function-oriented and relies heavily on functional abstraction and functional decomposition.
- ➔ During design, structured design methodology aims to control and influence implementing the design would have a hierarchical structure, with functionally cohesive modules and as few interconnections between modules as possible.
- ➔ Factoring is the process of decomposing the module so that the bulk of its work is done by its subordinates. There are four major steps in this strategy**.**

- **Restate the problem as a data flow diagram:**
    - ➔ To use the structure design methodology, the first step is to construct the data flow diagram for the problem.
    - ➔ There is a fundamental difference between DFDs drawn during requirement analysis and a DFD is drawn to model the problem domain.
    - ➔ The DFD during design represents how the data wall flows in the system when built.
    - ➔ The DFD shows the major transforms that the software will have and how the data will flow through different transforms so drawing a DFD for design is a very creative activity in which the designer visualizes the system and its processes and data flow.
    - ➔ The general rules for drawing DFD will be the same.

- **Identify the input and output data elements:**
    - ➔ Most systems have some basic, transformations that perform the required operations However, in most cases the transformation cannot be easily applied to the actual physical input and produce the desired physical output Instead, the put is first converted into a form on which the transformation can be applied with case Similarly, the main Transformation modules often produce outputs that have to be converted into the desired physical output.
    - ➔ The goal of this second step is to separate the transforms in the data flow diagram that convert the input or output to the desired format from the ones that perform the actual transformations.
    - ➔ For this separation, once the data flow diagram is ready, the next steps are to identify the highest abstract level of input and output.
    - ➔ The most abstract input data elements are those data elements in the data flow diagram that are farthest removed from the physical inputs but can still be considered inputs to the system.
    - ➔ The most abstract input data elements often have little resemblance to the actual physical data.
    - ➔ These are often the data elements obtained after operations like error checking, data validation, proper formatting and conversion are complete. Similarly, we identify the most abstract output data elements by starting from the outputs in the data flow diagram and traveling toward the inputs.
    - ➔ These are the data elements that are most removed from the actual outputs but can still be considered outgoing.

- **First-Level Factoring:**
    - ➔ Having identified the central transforms and the most abstract input - output data items, we are ready to identify some modules for the system. We first specify a main module; whose purpose is to invoke the subordinates.
    - ➔ The main module is therefore a coordinate module. For each of the most abstract input data items, an immediate subordinate module to the main module is specified.
    - ➔ Each of these modules is an input module, whose purpose is to deliver to the main module the most abstract data item for which it is created. Similarly, for each most abstract output data item, a subordinate module that is an output module that accepts data from the main module is specified.
    - ➔ Each of the arrows connecting these input and output subordinate modules is labeled with the respective abstract data item flowing in the proper direction.
    - ➔ Finally, for each central transform, a module subordinate to the main one is

specified.

- ➔ These modules will be transformed modules, whose purpose is to accept data from the main module, and then return the appropriate data back to the main module.
- ➔ The first-level factoring is straightforward, after the most abstract input and output data items are identified in the data flow diagram.
- ➔ The main module is the overall control module which will form the main program or procedure in the implementation of the design.
- ➔ Its ordinary module that invokes the input modules to get the most abstract data items, passes these to the appropriate transform modules, and delivers the results of the transform modules to other transform modules until the most abstract data items are obtained.

- **<u>Factoring of input, output, and transform branches:</u>**
  - ➔ The first-level factoring results in a very high-level structure, where each subordinate module has a lot of processing to do.
  - ➔ To simplify these modules, they must be factored into subordinate modules that will distribute the work of a module.
  - ➔ Each of the input, output, and transformation modules must be considered for factoring.
  - ➔ Let us start with the input modules.
  - ➔ The purpose of an input module, as viewed by the main program, is to produce some data. To factor an input module, the transform in the data flow diagram that produced the data item is now treated as a central transform. A subordinate input module is created for each input data stream coming into this new central transform, and a subordinate transform module is created for the new central transform.
  - ➔ Factoring of input modules will usually not yield any output subordinate modules. If the data flow diagram of the problem is sufficiently detailed, factoring of the input and output modules is straightforward. However, there are no such rules for factoring the central transforms.
  - ➔ The goal is to determine sub-transforms that will together compose the overall transform and then repeat the process for the newly found transforms, until we reach the atomic modules.
  - ➔ One way to factor a transform module is to treat it as a problem in its own right and start with a data flow diagram for it.
  - ➔ The inputs to the data flow diagram are the data coming into the module and the outputs are the data being returned by the module Each transform in this

data flow diagram represents a sub-transform of this transform.

➔ The central transform can be factored by creating a subordinate transform module for each of the transforms in this data flow diagram. This process can be repeated for the new transform modules that are created, until we reach atomic Modules.

Design Notation and Specification (Structure Chart)

Design notations are used during the process of design and are used to represent design or design decisions Once the designer is satisfied with the design he has produced, the design is to be precisely specified in the form of a document. The aim of design methodologies is not to reduce the process of design to a sequence of mechanical steps but to provide guidelines to aid the designer during the design process.

Structure Charts:

➔ Graphical design notations are frequently used during the design process to represent design or design decisions, so the design can be communicated to stakeholders in a succinct manner and evaluated For function-oriented design, the design can be represented graphically by structure charts.

➔ A Structure chart is a hierarchy Diagram.

➔ The structure of a program is made up of the modules of that program together with the interconnections between modules Every computer program has a structure, and given a program its structure can be determined.

➔ The structure chart of a program is a graphic representation of its structure. In a structure chart a module is represented by a box with the module name written in the box.

➔ **An arrow** from module A to module B represents that module A invokes module B. B is called the sub-ordinate of A, and A is called the super-ordinate of B. The arrow is labeled by the parameters received by B as input and the parameters returned by B as output, with the direction of flow of the input and output parameters represented by small arrows.

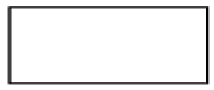
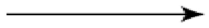
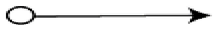
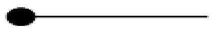
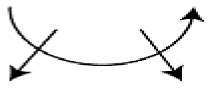
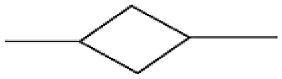
➔ In general, procedural information is not represented in a structure chart, and the focus is on representing the hierarchy of modules. However, there are situations where the designer may wish to communicate certain procedural information explicitly, like major loops and decisions. Such information can also be represented in a structure chart.

Structured Chart is a graphical representation which shows:

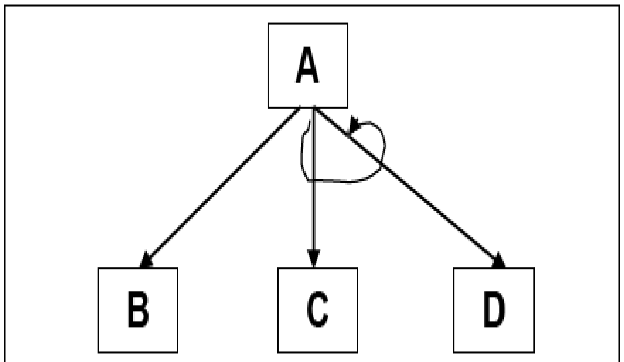
- System partitions into modules

- Hierarchy of component modules
- The relation between processing modules
- Interaction between modules
- Information passed between modules

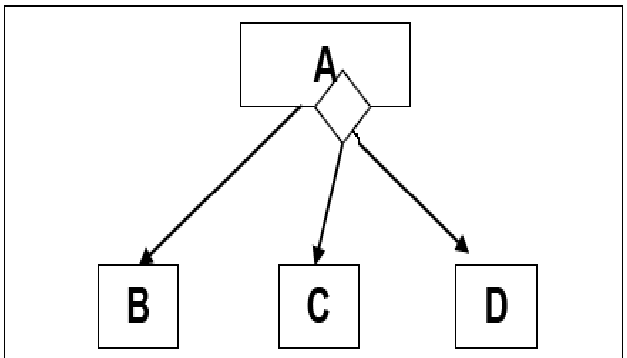
The following notations are used in structured chart:

SYMBOL	DESCRIPTION
	Module
	Arrow
	Data couple
	Control Flag
	Loop
	Decision

For example, let us consider a situation where module A has subordinates B, C, and D and A repeatedly calls the modules C and D.
This can be represented by a looping arrow around the arrows joining the subordinates C and D to A, as shown in Figure.
Major decisions can be represented similarly. For example, if the invocation of modules C and D in module A depends on the outcome of some decision, that is presented by a small diamond in the box for A, with the arrows joining C and D coming out of this diamond, as shown in Figure.



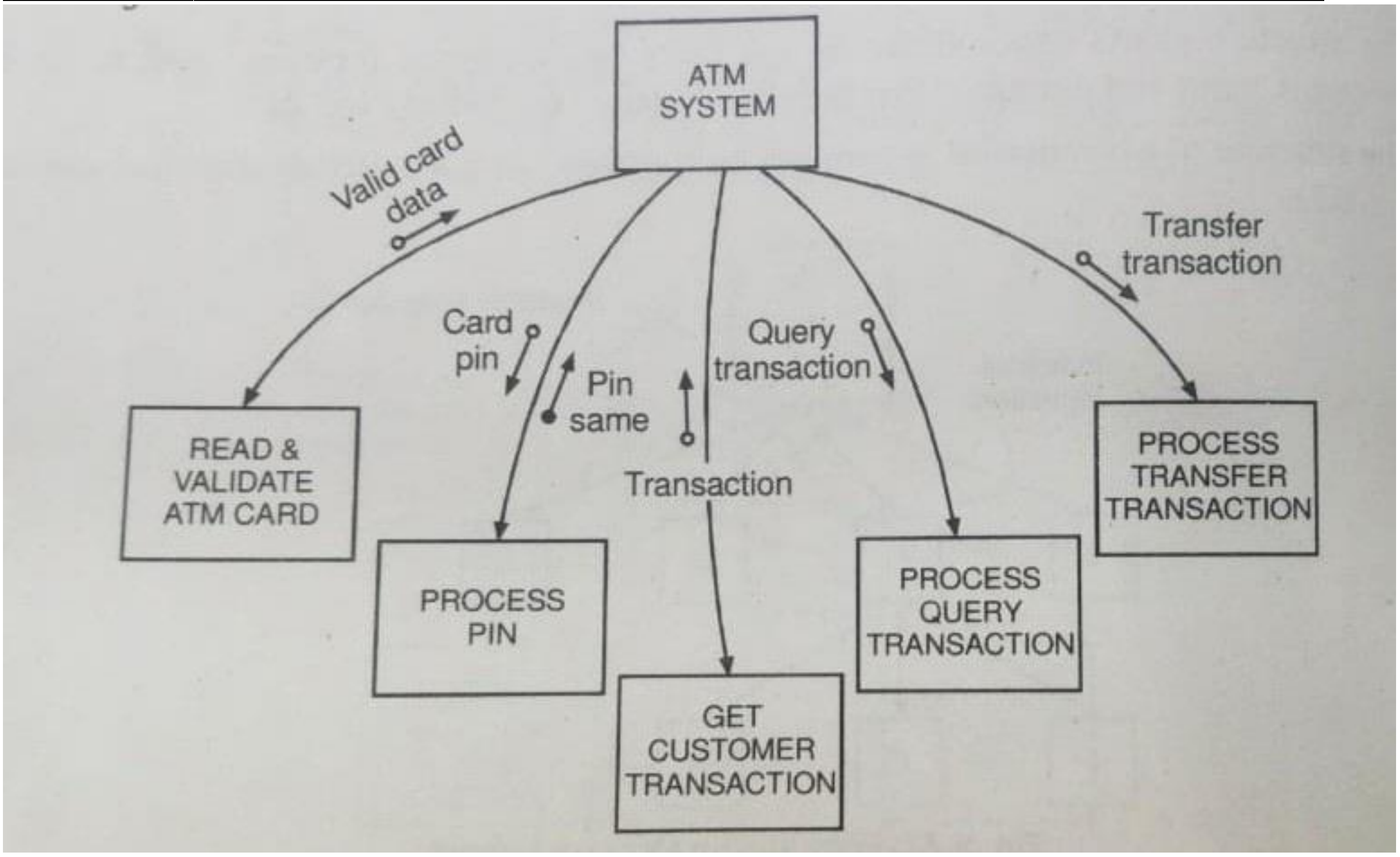
[Iteration



[Decision Making]

- ➔ Then there are modules that exist solely for the sake of transforming data into some other form.
- ➔ Such a module is called a transform module. Most of the computational modules typically fall in this category. Finally, there are modules whose primary concern is managing the flow of data to and from different subordinates. Such modules are called coordinate modules.
- ➔ A module can perform functions of more than one type of module. A structure chart is a nice representation for a design that uses functional abstraction.
- ➔ It shows the modules and their call hierarchy, the interfaces between the modules, and what information passes between modules.
- ➔ So, for a software system, once its structure is decided, the modules and their interfaces and dependencies get fixed.

Specifications:
In the design specification, a formal definition of these data structures should be given Module specification is the major part of the system design specification All modules in the system should be identified when the system design is complete, and these modules should be specified in the document During the system design only the module specification is obtained, because the internal details of the modules are defined later. To specify a module, the design document must specify the interface of the module, the abstract behavior of the module and the list of the all the modules used by the particular module The information is quite useful in maintaining and understanding the design.



Functional Approach vs. Object Oriented Approach

COMPARISON FACTORS	FUNCTION ORIENTED DESIGN	OBJECT ORIENTED DESIGN
Abstraction	The basic abstractions, which are given to the user, are real world functions.	The basic abstractions are not the real world functions but are the data abstraction where the real world entities are represented.
Function	Functions are grouped together by which a higher level function is obtained.	Functions are grouped together on the basis of the data they operate since the classes are associated with their methods.
execute	carried out using structured analysis and structured design i.e, data flow diagram	Carried out using UML
State information	In this approach the state information is often represented in a centralized shared memory.	In this approach the state information is not represented in a centralized memory but is implemented or distributed among the objects of the system.
Approach	It is a top down approach.	It is a bottom up approach.
Begins basis	Begins by considering the use case diagrams and the scenarios.	Begins by identifying objects and classes.

COMPARISON FACTORS	FUNCTION ORIENTED DESIGN	OBJECT ORIENTED DESIGN
Abstraction	The basic abstractions, which are given to the user, are real world functions.	The basic abstractions are not the real world functions but are the data abstraction where the real world entities are represented.
Decompose	In function-oriented design we decompose in function/procedure level.	We decompose at class level.
Use	This approach is mainly used for computation sensitive applications.	This approach is mainly used for evolving systems which mimics a business or business case.